Decoding billions of integers per second through vectorization

D. Lemire[1]*, L. Boytsov[2]

[1]*LICEF Research Center, TELUQ, Montreal, QC, Canada*
[2]*Carnegie Mellon University, Pittsburgh, Pennsylvania, USA*

SUMMARY

In many important applications—such as search engines and relational database systems—data is stored in the form of arrays of integers. Encoding and, most importantly, decoding of these arrays consumes considerable CPU time. Therefore, substantial effort has been made to reduce costs associated with compression and decompression. In particular, researchers have exploited the superscalar nature of modern processors and SIMD instructions. Nevertheless, we introduce a novel vectorized scheme called SIMD-BP128* that improves over previously proposed vectorized approaches. It is nearly twice as fast as the previously fastest schemes on desktop processors (varint-G8IU and PFOR). At the same time, SIMD-BP128* saves up to 2 bits per integer. For even better compression, we propose another new vectorized scheme (SIMD-FastPFOR) that has a compression rate within 10% of a state-of-the-art scheme (Simple-8b) while being two times faster during decoding.

KEY WORDS: performance; measurement; index compression; vector processing

1. INTRODUCTION

Computer memory is a hierarchy of storage devices that range from slow and inexpensive (disk or tape) to fast but expensive (registers or CPU cache). In many situations, application performance is inhibited by access to slower storage devices, at lower levels of the hierarchy. Previously, only disks and tapes were considered to be slow devices. Consequently, application developers tended to optimize only disk and/or tape I/O. Nowadays, CPUs have become so fast that access to main memory is a limiting factor for many workloads [1, 2].

Data compression helps to load and keep more of the data into a faster storage. Hence, high speed compression schemes can improve the performances of database systems [3, 4, 5] and text retrieval engines [6, 7, 8, 9, 10].

We focus on compression techniques for 32-bit integer sequences. It is best if most of the integers are small, because we can save space by representing small integers more compactly, i.e., using short codes. Assume, for example, that none of the values is larger than 255. Than we can encode each integer using one byte, thus, achieving a compression rate of 4: an integer uses 4 bytes in the uncompressed format.

In relational database systems, column values are transformed into integer values by dictionary coding [11, 12, 13, 14, 15]. To improve compressibility, we may map the most frequent values to the smallest integers [16]. In text retrieval systems, word occurrences are commonly represented by sorted lists of integer document identifiers, also known as posting lists. These identifiers are

*Correspondence to: LICEF Research Center, TELUQ, 100 Sherbrooke West Montreal, QC, H2X 3P2.

Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada; contract/grant number: 261437

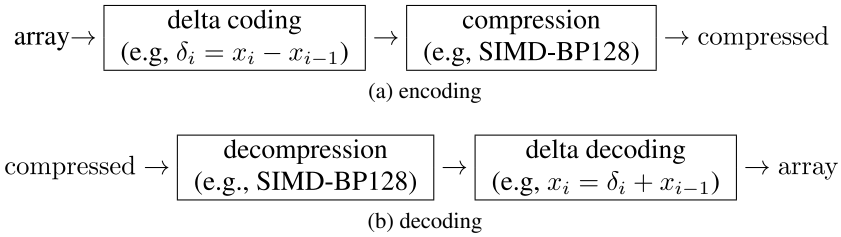


Figure 1. Encoding and decoding of integer arrays using delta coding and an integer compression algorithm

converted to small integer numbers through data differencing. Other database indexes can also be stored similarly [17].

A mainstream approach to data differencing in text retrieval systems is delta coding (see Fig. 1). Instead of storing the original array of sorted integers ($x_1, x_2, \ldots$ with $x_i \leq x_{i+1}$ for all i), we keep only the difference between successive elements together with the initial value: ($x_1, \delta_2 = x_2 - x_1, \delta_3 = x_3 - x_2, \ldots$). The differences (or deltas) are non-negative integers that are typically much smaller than the original integers. Therefore, they can be compressed more efficiently We can then reconstruct the original arrays by computing prefix sums ($x_j = x_1 + \sum_{i=2}^{j} \delta_j$).

An engineer might be tempted to compress the result using generic compression tools such as LZO, Google Snappy, FastLZ, LZ4 or gzip. Yet this might be ill-advised. Our fastest schemes are an order of magnitude faster than a fast generic library like Snappy, while compressing better (see § 6.5).

Instead, it might be preferable to compress these arrays of integers using specialized schemes based on Single-Instruction, Multiple-Data (SIMD) operations. Stepanov et al. [9] reported that their SIMD-based varint-G8IU algorithm outperformed the classic variable byte coding method (see § 4.4) by 300%. They also showed that use of SIMD instructions allows one to improve performance of decoding algorithms by more than 50%.

In Table I, we report the speeds of the most efficient decoding algorithms described in the literature as well as the best speed we obtained on desktop processors. To account for different processor speeds, we also express the processing time in the number of CPU cycles per integer. We report our own speed in a conservative manner: (1) our timings are based on the wall-clock time and not the commonly used CPU time, (2) our timings incorporate all of the decoding operations including the computation of the prefix sum whereas this is sometimes omitted by other authors [18], (3) we report a speed of 2300 million integers per second (mis) achievable for realistic data sets, while higher speed is possible (e.g., we report a speed of 2500 mis on some realistic data and 2800 mis on some synthetic data).

From Table I one can gather that varint-G8IU—which can be viewed as an improvement on the Group Varint Encoding [10] (varint-GB) used by Google—is, probably, the fastest method (except for our new schemes) in the literature. Yet these numbers should be compared with care since hardware, benchmarking methodology, and data sets differ. According to our own experimental evaluation (see Tables IV, V and Fig. 10), varint-G8IU is, indeed is one of the most efficient methods, but there are previously published schemes that offer similar or even slightly better performance (for some data). We, in turn, were able to further surpass the decoding speed of varint-G8IU by a factor of two while improving the compression rate.

For most schemes, the prefix sum computation is so fast as to represent 20% or less of the running time. However, because our novel schemes are much faster, the prefix sum can account for the majority of the running time.

Hence, we had to experiment with faster alternatives. We find that a vectorized prefix sum using SIMD instructions can be twice as fast. Without vectorized delta coding, we were unable to reach a speed of two billion integers per second.

Table I. Recent best decoding speeds in millions of 32-bit integers per second (mis) reported for integer compression on realistic data. We also report the best number of CPU cycles per integer during decoding.

	Speed	Cycles/int	Fastest scheme	Processor
this paper	**2300**	**1.5**	**SIMD-BP128***	Core i7 (3.4 GHz)
Stepanov et al. (2011) [9]	1512	2.2	varint-G8IU	Xeon (3.3 GHz)
Anh and Moffat (2010) [19]	1030	2.3	binary packing	Xeon (2.33 GHz)
Silvestri and Venturini (2010) [18]	835	—	VSEncoding	Xeon
Yan et al. (2009) [7]	1120	2.4	NewPFD	Core 2 (2.66 GHz)
Zhang et al. (2008) [20]	890	3.6	PFOR2008	Pentium 4 (3.2 GHz)

2. FAST DELTA CODING AND DECODING

All of the schemes we consider rely on delta coding over 32-bit unsigned integers. Delta coding transforms a sorted array into an array of differences between nearby elements. Computation of deltas is typically considered a trivial operation, which accounts for a small fraction of total decoding time. Consequently, authors do not discuss it in details. In our experience, a straightforward implementation of delta decoding can be four times slower than the decompression of small integers.

We have implemented and evaluated three approaches to data differencing:

1. The standard form of delta coding is simple and requires merely one subtraction per value during encoding ($\delta_i = x_i - x_{i-1}$) and one addition per value during decoding to effectively compute the prefix sum ($x_i = \delta_i + x_{i-1}$).
2. A modified delta encoding includes an additional subtraction by one: $\delta_i = x_i - x_{i-1} - 1$. In that, the decoding part requires an extra addition: $x_i = \delta_i + x_{i-1} + 1$. This approach generates non-negative deltas when we have sorted sequences of *distinct* integers ($x_i < x_{i+1}$ for all i).
3. A vectorized delta encoding leaves the first four elements unmodified. From each of the remaining elements with index i, we subtract the element with the index $i - 4$: $\delta_i = x_i - x_{i-4}$. In other words, the original array $(x_1, x_2, \ldots)$ is converted into $(x_1, x_2, x_3, x_4, \delta_5 = x_5 - x_1, \delta_6 = x_6 - x_2, \delta_7 = x_7 - x_3, \delta_8 = x_8 - x_4, \ldots)$. An advantage of this approach is that we can compute four differences using a single SIMD operation. This operation carries out an element-wise subtraction for two four-element *vectors*. The decoding part is symmetric and involves the addition of the element x_{i-4}: $x_i = \delta_i + x_{i-4}$. Again, we can use a single SIMD instruction to carry out four additions simultaneously.

Using the second approach, we cannot reconstruct the values from the deltas faster than ≈1250 mis or 2.7 cycles/int. In contrast, we can get a speed of ≈2000 mis or 1.7 cycles/int with the standard delta decoding (the first approach) by manually unrolling the loops. Thus, we proceeded with common delta coding (which does not subtract one).

Clearly, it is impossible to decode compressed integers at a rate of 2 or 3 billion integers per second if the computation of the prefix sum itself runs at 2 billion integers per second. Hence, we implemented a vectorized version of delta coding. Vectorized delta decoding is much faster (≈5000 mis vs. ≈2000 mis). However, it comes at a price: vectorized deltas are, on average, four times larger which increases the storage cost by up to 2 bits (e.g, see Table V).

Because memory bandwidth may become a bottleneck [1], we prefer to compute delta coding and decoding in place. To this end, we compute deltas in decreasing index order, starting from the largest index. In contrast, the delta decoding proceeds in increasing index order, starting from the beginning of the array. Further, our implementation requires two passes: one pass to reconstruct the deltas from their compressed format and another pass to compute the prefix sum (§ 6.2). To improve data locality and reduce cache misses, arrays containing more than 2^{16} integers (256 KB) are broken down into smaller arrays and each array is decompressed independently. Experiments with synthetic data have showed that reducing cache misses in this manner leads to more than *a twofold improvement* in decoding speed for some schemes without degrading the compression rate.

```
struct Fields4_8  {
    unsigned    Int1: 4;
    unsigned    Int2: 4;
    unsigned    Int3: 4;
    unsigned    Int4: 4;
    unsigned    Int5: 4;
    unsigned    Int6: 4;
    unsigned    Int7: 4;
    unsigned    Int8: 4;
};
```

```
struct Fields5_8 {
  unsigned    Int1: 5;
  unsigned    Int2: 5;
  unsigned    Int3: 5;
  unsigned    Int4: 5;
  unsigned    Int5: 5;
  unsigned    Int6: 5;
  unsigned    Int7: 5;
  unsigned    Int8: 5;
};
```

Figure 2. Eight bit-packed integers represented as two structures in C/C++. Integers in the left panel use 4-bit fields, while integers in the right panel use 5-bit fields.

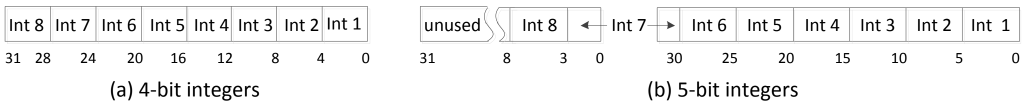


Figure 3. Example of two bit-packed representations of 8 small integers. For convenience, we indicate a starting bit number for each field (numeration begins from zero). Integers in the left panel use 4-bit each and, consequently, they fit into a single 32-bit word. Integers in the right panel use 5-bit each. The complete representation uses two 32-bit words: 24-bits are unoccupied.

3. FAST BIT UNPACKING

Bit packing is a process of encoding small integers in $[0, 2^b)$ using b bits each: b can be arbitrary and not just 8, 16, 32 or 64. Each number is written using a string of exactly b bits. Bit strings of fixed size b are concatenated together into a single bit string, which can span several 32-bit words. If some integer is too small to use all b bits, it is padded with zeros.

Languages like C and C++ support the concept of bit packing through bit fields. An example of two C/C++ structures with bit fields is given in Fig. 2. Each structure in this example stores 8 small integers. The structure `Fields4_8` uses 4 bits per integer ($b = 4$), while the structure `Fields5_8` uses 5 bits per integer ($b = 5$).

Assuming that bit fields in these structures are stored compactly, i.e., without gaps, and the order of the bit fields is preserved, the 8 integers are stored in the memory as shown in Fig. 3. If any bits remain unused, their values can be arbitrary. All small integers on the left panel in Fig. 3 fit into a single 32-bit word. However, the integers on the right panel require two 32-bit words with 24 bits remaining unused (these bits can be arbitrary). The field of the 7^{th} integer crosses the 32-bit word boundary: the first two bits use bits 30–31 of the first words, while the remaining three bits occupy bits 0–2 of the second word (bits are enumerated starting from zero).

Unfortunately, language implementers are not required to ensure that the data is fully packed. Most importantly, they do not have to provide packing and unpacking routines that are optimally fast. Hence, we implemented bit packing and unpacking using our own procedures as proposed by Zukowski et al. [21]. In Fig. 4, we give C/C++ implementations of such procedures assuming that fields are laid out as depicted in Fig. 3. The packing procedures can be implemented similarly and we omit them for simplicity of exposition.

In some cases, we use bit packing even though some integers are larger than $2^b - 1$ (see § 4.8). In effect, we want to pack only the first b bits of each integer, which can be implemented by applying a bit-wise logical `and` operation with the mask $2^b - 1$ on each integer. These extra steps slow down the bit packing (see § 6.3).

The procedure `unpack4_8` decodes eight 4-bit integers. Because these integers are tightly packed, they occupy exactly one 32-bit word. Given that this word is already loaded in a register, each integer can be extracted using at most four simple operations (shift, mask, store, and pointer increment). Unpacking is efficient because it does not involve any branching.

```
void unpack4_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))        & 15;
  *out++ = ((*in) >> 4 )  & 15;
  *out++ = ((*in) >> 8 )  & 15;
  *out++ = ((*in) >> 12)  & 15;
  *out++ = ((*in) >> 16)  & 15;
  *out++ = ((*in) >> 20)  & 15;
  *out++ = ((*in) >> 24)  & 15;
  *out =   ((*in) >> 28);
}
```

```
void unpack5_8(const uint32_t* in,
          uint32_t* out) {
  *out++ = ((*in))        & 31;
  *out++ = ((*in) >> 5 )  & 31;
  *out++ = ((*in) >> 10)  & 31;
  *out++ = ((*in) >> 15)  & 31;
  *out++ = ((*in) >> 20)  & 31;
  *out++ = ((*in) >> 25)  & 31;
  *out = ((*in) >> 30);
  ++in;
  *out++ |= ((*in) & 7) << 2;
  *out =    ((*in) >> 3) & 31;
}
```

Figure 4. Two procedures to unpack eight bit-packed integers. The procedure `unpack4_8` works for $b = 4$ while procedure `unpack5_8` works for $b = 5$. In both cases we assume that (1) integers are packed tightly, i.e., without gaps, (2) packed representations use whole 32-bit words: values of unused bits are undefined.

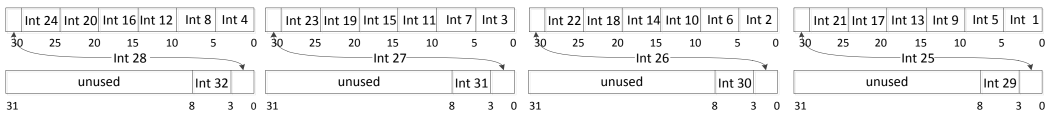


Figure 5. Example of bit-packed representations of 32 small integers used with SIMD-based, i.e., vectorized, packing/unpacking. For convenience, we show a starting bit number for each field (numeration begins from zero). Integers use 5-bit each. Words in the second row follow (i.e., have larger addresses) words of the first row. Curved lines with arrows indicate that integers 25–28 are each split between two words.

The procedure `unpack5_8` decodes eight 5-bit integers. This case is more complicated, because the packed representation uses two words: the field for the 7$^{\text{th}}$ integer crosses word boundaries. The first two (lower order) bits of this integer are stored in the first word, while the remaining three (higher order) bits are stored in the second word. Decoding does not involve any branches and most integers are extracted using four simple operations.

Decoding routines `unpack4_8` and `unpack5_8` operate on *scalar* 32-bit values. An effective way to improve performance of these routines involves *vectorization* [11, 22]. Consider listings in Fig. 4 and assume that `in` and `out` are pointers to m-element vectors instead of scalars. Further, assume that scalar operators (shifts, assignments, and bit-wise logical operations) are vectorized. For example, a bit-wise shift is applied to all m vector elements at the same time. Then, a single call to `unpack5_8` or `unpack4_8` decodes $m \times 8$ rather than just eight integers.

Recent x86 processors have SIMD instructions that operate on vectors of four 32-bit integers ($m = 4$) [23, 24, 25]. We can use these instructions to achieve a better decoding speed. A sample SIMD-based data layout for $b = 5$ is given in Fig. 5. Integers are divided among series of four 32-bit words in a round-robin fashion. When a series of four words overflows, the data *spills* over to the next series of 32-bit integers. In this example, the first 24 integers are stored in the first four words (the first row in Fig. 5), integers 25–28 are each split between different words, and the remaining integers 29–32 are stored in the second series of words (the second row of the Fig. 5).

These data can be processed using a vectorized version of the procedure `unpack5_8`, which is obtained from `unpack5_8` by replacing scalar operations with respective SIMD instructions that operate on four 32-bit vectors. In the beginning of such a procedure the pointer `in` points to the first 128-bit chunk of data displayed in row one of the Fig. 5. The first shift-and-mask operation extracts 4 small integers at once. Then, these integers are written to the target buffer using a *single* 128-bit SIMD store operation. The shift-and-mask is repeated until we extract the first 24 numbers and the first two bits of the integers 25–28. At this point the unpack procedure increases the pointer `in` and loads the next 128-bit chunk into a register. Using an additional mask operation, it extracts the remaining 3 bits of integers 25–28. These bits are combined with already obtained first 2 bits

(for each of the integers 25–28). Finally, we store integers 25–28 and finish processing the second 128-bit chunk by extracting numbers 29–32.

Our vectorized data layout is interleaved. That is, the first four integers (Int 1, Int 2, Int 3, and Int 4 in Fig. 5) are packed into 4 different 32-bit words. The first integer is immediately adjacent to the fifth integer (Int 5). Schlegel et al. [26] called this model *vertical*. Instead we could ensure that the integers are packed sequentially (e.g. Int 1, Int 2, and Int 3 could be stored in the same 32-bit word). Schlegel et al. called this alternative model *horizontal* and it is used by Willhalm et al. [22] (see § 4.6.1). One potential benefit of a sequential layout is that random access to a few integers could be faster since fewer words need to be accessed.

4. RELATED WORK

Some of the earliest integer compression techniques are Golomb coding [27], Rice coding [28], as well as Elias gamma and delta coding [29]. In recent years, several faster techniques have been added such as the Simple family, binary packing, and patched coding. We briefly review them and describe our implementations.

Because we work with unsigned integers, we make use of two representations: binary and unary. In both systems numbers are represented using only two digits: 0 and 1. The binary notation is a standard positional base-2 system (e.g., $0 \to 0$, $1 \to 1$, $2 \to 10$, $3 \to 11$). In unary notation, we represent a number x as a sequence of x digits 1 followed by the digit 0 (e.g., $0 \to 0$, $1 \to 10$, $2 \to 110$, $3 \to 1110$). If the number x is known to be always non-zero, we can store $x - 1$ instead for better compression.

4.1. Golomb and Rice coding

In Golomb coding [27], given a fixed parameter b and an integer v to be compressed, the quotient $\lfloor v/b \rfloor$ is coded in unary. The remainder $r = v \bmod b$ is stored using the usual binary notation with no more than $\lceil \log_2 b \rceil$ bits. When b is chosen to be a power of two, the resulting algorithm is called Rice coding [28]. The parameter b can be chosen optimally by assuming some that the integers follow a known distribution [27].

Unfortunately, Golomb and Rice coding are much slower than a simple scheme such as Variable Byte [6, 7, 30] (see § 4.4) which, itself, falls short of our goal of decoding billions of integers per second (see § 6.4–6.5).

4.2. Interpolative coding

If speed is not an issue but high compression over sorted arrays is desired, interpolative coding [31] might be appealing. In this scheme, we first store the lowest and the highest value, x_1 and x_n, e.g., in a uncompressed form. Then a value in-between is stored in a binary form, using the fact this value must be in the range (x_1, x_n). For example, if $x_1 = 16$ and $x_n = 31$, we know that for any value x in between, the difference $x - x_1$ is from 0 to 15. Hence, we can encode this difference using only 4 bits. The technique is then repeated recursively. Unfortunately, it is slower than Golomb coding [7, 6].

4.3. Elias gamma and delta coding

An Elias gamma code [29, 32] consists of two parts. The first part encodes in unary notation the minimum number of bits necessary to store the integer in binary notation ($\lceil \log_2(x + 1) \rceil$). The second part represents the integer in binary notation less the most significant digit. If the integer is equal to zero or one, the second part is empty (e.g., $0 \to 0$, $1 \to 10$, $2 \to 110\,0$, $3 \to 110\,1$, $4 \to 1110\,00$). If integers are non-zero, we can code their values decremented by one to improve compression further. As numbers become large, gamma codes become inefficient. For better compression, Elias delta codes encode the first part (the number $\lceil \log_2(x + 1) \rceil$) using the Elias gamma code, while the second part is coded in binary notation as before. For example, to code

the number 8 using the Elias delta code, we must first store $4 = \lceil \log_2(8+1) \rceil$ as a gamma code (1110 00) and then we can store all but the most significant bit of the number 8 in binary notation (000). The net result is 1110 00 000.

However, Variable Byte is twice as fast as Elias gamma and delta coding [18]. Hence, like Golomb coding, it falls short of our objective of compressing billions of integers per second.

4.3.1. k-gamma However, to ease vectorization, the data is stored in blocks of k integers using the same number of bits where $k \in \{2, 4\}$. (This approach is similar to binary packing described in § 4.6.) As with regular gamma coding, we use unary codes to store this number of bits though we only have one such number for k integers.

The binary part of the gamma codes are stored using the same vectorized layout we described in § 3 (known as vertical or interleaved). During decompression, we decode integer in groups of k integers. For each group we first retrieve the binary length from a gamma code. Then, we decode group elements using a sequence of mask-and-shift operations similar to the fast bit unpacking technique we described in § 3. This step does not require branching.

Schlegel et al. report best decoding speeds of ≈550 mis (≈2100 MB/s) on synthetic data using an Intel Core i7-920 processor (2.67 GHz). These results fall short of our objective to compression billions of integers per second.

4.4. *Variable Byte and byte-oriented encodings*

Variable Byte is a popular technique that is known under many names (v-byte, var-byte, vbyte, varint, VInt, VB [9] or Escaping [30]). Variable Byte codes the data in units of bytes: it uses the lower-order seven bits to store the data, while the eighth bit is used as an implicit indicator of a code length. Namely, the eighth bit is equal to 1 only for the last byte of a sequence that encodes an integer. For example:

- Integers in $[0, 2^7)$ are written using one byte: The first 7 bits are used to store the binary representation of the integer and the eighth bit is set to 1.
- Integers in $[2^7, 2^{14})$ are written using two bytes, the eighth bit of the first byte is set to 0 whereas the eighth bit of the second byte is set to 1. The remaining 14 bits are used to store the binary representation of the integer.

For a concrete example, consider the number 200. It is written as 11001000 in binary notation. Variable Byte would code it using 16 bits as 10000001 01001000.

When decoding, bytes are read one after the other: we discard the eighth bit if it is zero, and we output a new integer whenever the eighth bit is one.

Though Variable Byte rarely compresses data optimally, it is reasonably efficient. In our tests, Variable Byte encodes data three times faster than most alternatives. Moreover, when the data is not highly compressible, it can match the compression rates of more parsimonious schemes.

Stepanov et al. [9] generalize Variable Byte into a family of byte-oriented encodings. Their main characteristic is that each encoded byte contains bits from only one integer. However, whereas Variable Byte uses one bit per byte as descriptor, alternative schemes can use other arrangements. For example, varint-G8IU [9] and Group Varint [10] (henceforth varint-GB) regroup all descriptors in a single byte. Such alternative layouts make easier the simultaneous decoding of several integers.

For example, varint-GB uses a single byte to describe 4 integers, dedicating 2 bits per integer. The scheme is better explained by an example. Suppose that we want to store the integers 2^{15}, 2^{23}, 2^7, and 2^{31}. In the usual binary notation, we would use 2, 3, 1 and 4 bytes, respectively. We can store the sequence as 2, 3, 1, 4 as 1, 2, 0, 3 if we assume that each number is encoded using a non-zero number of bytes. Each one of these 4 integers can be written using 2 bits (as they are in {0,1,2,3}). We can pack them into a single byte containing the bits 01,10,00, and 11. Following this byte, we write the integer values using $2 + 3 + 1 + 4 = 10$ bytes.

Whereas varint-GB codes a fixed number of integers (4) using a single descriptor, varint-G8IU uses a single descriptor for a group of 8 bytes, which represent compressed integers. Each 8-byte group may store from 2 to 8 integers. A single-byte descriptor is placed immediately before this

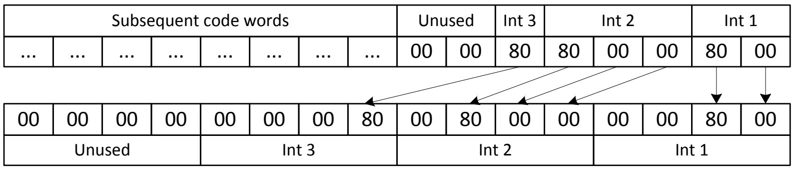


Figure 6. Example of simultaneous decoding of 3 integers in the scheme varint-G8IU using the shuffle instruction. The integers 2^{15}, 2^{23}, 2^7 are packed into the 8-byte block with 2 bytes being unused. Byte values are given by *hexadecimal* numbers. The target 16-byte buffer bytes are either copied from the source 16-byte buffer or are filled with zeros. Arrows indicate which bytes of the source buffer are copied to the target buffer as well as their location in the source and target buffers.

8-byte group. Each bit in the descriptor represents a single data byte. Whenever a descriptor bit is set to 0, then the corresponding byte is the end of an integer. This is symmetrical to the Variable Byte scheme described above, where the descriptor bit value 1 denotes the last byte of an integer code.

In the example we used for varint-GB, we could only store the first 3 integers (2^{15}, 2^{23}, 2^7) into a single 8-byte group, because storing all 4 integers would require 10 bytes. These integers use 2, 3, and 1 bytes, respectively, whereas the descriptor byte is equal to 11001101 (in the binary notation). The first two bits (01) of the descriptor tell us that the first integer uses 2 bytes. The next three bits (011) indicate that the second integer requires 3 bytes. Because the third integer uses a single byte, the next (sixth) bit of the descriptor would be 0. In this model, the last two bytes cannot be used and, thus, we would set the last two bits to 1.

On most recent x86 processors, integers packed with varint-G8IU can be efficiently decoded using the SSSE3 *shuffle* instruction: `pshufb`. This assembly operation selectively copies byte elements of a 16-element vector to specified locations of the target 16-element buffer and replaces selected elements with zeros.

The name "shuffle" is a misnomer, because certain source bytes are omitted, while others may be copied multiple times to a number of different locations. The operation takes two 16 element vectors (of $16 \times 8 = 128$ bits each): the first vector contains the bytes to be shuffled into an output vector whereas the second vector serves as a *shuffle mask*. Each byte in the shuffle mask determines which value will go in the corresponding location in the output vector. If the last bit is set (that is, if the value of the byte is larger than 127), the target byte is zeroed. For example, if the shuffle mask contains the byte values $127, 127, \ldots, 127$, then the output vector will contain only zeros. Otherwise, the first 4 bits of the i^{th} mask element determine the index of the byte that should be copied to the target byte i. For example, if the shuffle mask contains the byte values $0, 1, 2, \ldots, 15$, then the bytes are simply copied in their original locations.

In Fig. 6, we illustrate one step of the decoding algorithm for varint-G8IU. We assume that the descriptor byte, which encodes lengths of integers 3 integers (2^{15}, 2^{23}, 2^7), is already retrieved. The value of the descriptor byte was used to obtain a proper shuffle mask for `pshufb`. This mask (which is precomputed before decoding starts) defines a sequence of operations that copy bytes from the source to the target buffer or fill selected bytes of the target buffer with zeroes. All these byte operations are carried out in parallel in the following manner (byte numeration starts from *zero*):

- The first integer uses only 2 bytes, which are both copied to bytes 0–1 of the target buffer Bytes 2–3 of the target buffer are zeroed.
- Likewise, we copy bytes 2–4 of the source buffer to bytes 4–6 of the target buffer. Byte 7 of the target buffer is zeroed.
- The last integer uses only one byte 5: we copy the value of this byte to byte 8 and zero bytes 9–11.
- The bytes 12–15 of the target buffer are currently unused and will be filled out by subsequent decoding steps. In the current step, we may fill them with arbitrary values, e.g., zeros.

Table II. Encoding mode for Simple-8b scheme. Between 1 and 240 integers are coded with one 64-bit word.

selector value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
integers coded	240	120	60	30	20	15	12	10	8	7	6	5	4	3	2	1
bits per integer	0	0	1	2	3	4	5	6	7	8	10	12	15	20	30	60

We do not know whether Google implemented varint-GB using SIMD instructions [10]. However, Schlegel et al. [26] and Popov [8] described the application of the `pshufb` instruction to accelerate decoding of a varint-GB scheme (which Schlegel et al. called *4-wise null suppression*).

Stepanov et al. [9] found varint-G8IU to compress slightly better than a SIMD-based varint-GB while being up to 20% faster. Compared to the common Variable Byte, varint-G8IU had a slightly worse compression rate (up to 10%), but it is 2–3 times faster.

4.5. *The Simple family*

Whereas Variable Byte takes a fixed input length (a single integer) and produces a variable-length output (1, 2, 3 or more bytes), at each step the Simple family outputs a fixed number of bits, but processes a variable number of integers, similar to varint-G8IU. However, unlike varint-G8IU, schemes from the Simple family are not byte-oriented. Therefore, they may fare better on highly compressible arrays (e.g., they could compress a sequence of numbers in $\{0, 1\}$ to $\approx$1 bit/int).

The most competitive Simple scheme on 64-bit processors is Simple-8b [19]. It outputs 64-bit words. The first 4 bits of every 64-bit word is a selector that indicates an encoding mode. The remaining 60 bits are employed to keep data. Each integer is stored using the same number of bits b. Simple-8b has 2 schemes to encode long sequences of zeros and 14 schemes to encode positive integers. For example:

- Selector values 0 or 1 represent sequences containing 240 and 120 zeros, respectively. In this instance the 60 data bits are ignored.
- The selector value 2 corresponds to $b = 1$. This allows us to store 60 integers having values in $\{0,1\}$, which are packed in the data bits.
- The selector value 3 corresponds to $b = 2$ and allows one to pack 30 integers having values in $[0, 4]$ in the data bits.

And so on (see Table II): the larger is the value of the selector, the larger is b, and the fewer integers one can fit in 60 data bits. During coding, we try successively the selectors starting with value 0. That is, we greedily try to fit as many integers as possible in the next 64-bit word.

Other schemes such as Simple-9 [6] and Simple-16 [7] use words of 32 bits. (Simple-9 and Simple-16 can also be written as S9 and S16 [7].) While these schemes may sometimes compress slightly better, they are generally slower. Hence, we omitted them in our experiments. Unlike Simple-8b that can encode integers in $[0, 2^{60})$, Simple-9 and Simple-16 are restricted to integers in $[0, 2^{28})$.

To minimize branching, we implemented Simple-8b using a C++ `switch case` that selects one of 16 functions, that is, one for each selector value. Such functions are faster because loop unrolling eliminates branching. (Anh and Moffat [19] referred to this optimization as *bulk unpacking*.)

While Simple-8b is not as fast as Variable Byte during encoding, it is still faster than many alternatives. Because the decoding step can be implemented efficiently (with little branching), we also get a good decoding speed while achieving a better compression rate than Variable Byte.

4.6. *Binary Packing*

Binary packing is closely related to Frame-Of-Reference (FOR) from Goldstein et al. [33] and tuple differential coding from Ng and Ravishankar [34]. In such techniques, arrays of values are partitioned into blocks (e.g., of 128 integers). In FOR, the range of values in the blocks is first coded and then all values in the block are written in reference to the range of values: for example, if the values in a block are integers in the range $[1000, 1127]$, then they can be stored using

7 bits per integer ($\lceil \log_2(1127 + 1 - 1000) \rceil = 7$) as offsets from the number 1000 stored in the binary notation. In our approach to binary packing, we assume that integers are small, so we only need to code a bit width b per block (to represent the range). Then, successive values are stored using b bits per integer using fast bit packing functions. Anh and Moffat called binary packing *PackedBinary* [19] whereas Delbru et al. [35] called their 128-integer binary packing FOR and their 32-integer binary packing AFOR-1.

Reading and writing unaligned data can be as fast as reading and writing aligned data on recent Intel processors—as long as we do not cross a 64-byte cache line. Nevertheless, we still wish to align data on 32-bit boundaries when using regular binary packing and on 128-bit boundaries when using vectorized binary packing. Hence, we implemented binary packing over blocks of 32-bit integers (henceforth BP32) by regrouping 4 blocks into a meta-block of 128 integers. The encoded representation of the meta-block is preceded by a descriptor. The descriptor is a 32-bit word that stores 4 bit widths b (8 bit per width). We also experimented with versions of binary packing on few integers (8 integers and 16 integers). Because these versions are slower, we omit them from our experiments.

We also implemented a vectorized binary packing over blocks of 128 integers (henceforth SIMD-BP128). We regroup 16 blocks into a meta-block of 2048 integers. As in BP32, the encoded representation of a meta-block is preceded by a 128-bit descriptor word keeping bit widths (8 bit per width). SIMD-BP128 employs vectorized bit packing whereas BP32 relies on the regular C++ bit packing as described in §3.

A key step during encoding is that we must determine the maximum of the integer logarithm of the integers ($\max_i \lceil \log_2(x_i + 1) \rceil$). If done naively, this step can take up most of the running time: the computation of the integer logarithm is slower than a fast operation such as a shift or an addition. Instead, we carry out a bit-wise logical `or` on all the integers and compute the integer logarithm of the result. This shortcut is possible due to the equation: $\max_i \lceil \log_2(x_i + 1) \rceil = \lceil \log_2 \vee_i (x_i + 1) \rceil$ where $\vee$ refers to the bit-wise logical `or`. As an additional optimization, we used the x86 `bsr` assembly instruction for computing the integer logarithm (as it provides $\lceil \log_2(x + 1) \rceil - 1$ whenever $x > 0$).

If the gaps between integers are similar, that is, there are few large gaps, binary packing can be have a good compression rate. Indeed, consider arrays made of b-bit integers selected uniformly at random (in $[0, 2^b)$). The Shannon entropy is b bits while the bit rate of SIMD-BP128 will be no more than $b + 1/16$ bits. In Appendix A, we derive a more general information theoretical lower bound on the compression rate of binary packing.

4.6.1. Horizontally vectorized binary packing Willhalm et al. [22] proposed a variant of binary packing that employs a sequential (or horizontal) layout as opposed to the interleaved (or vertical) that we use for SIMD-BP128. In their scheme, decoding relies on the SSSE3 shuffle operation `pshufb` (like varint-G8IU). After we determine the bit width b of integers in the block, one decoding step typically includes the following operations:

1. Loading data into the source 16-byte buffer (this step may require a 16-byte alignment).
2. Distributing 3–4 integers stored in the source buffer among four 32-bit words of the target buffer. This step, which requires loading a shuffle mask, is illustrated by Fig. 7 (for 5-bit integers). Note that unlike varint-G8IU, the integers in the source buffer are not necessarily aligned by byte boundaries (unless b is 8, 16, or 32). Hence, after the shuffle operation, (1) the integers copied the target buffer may not be aligned on boundaries of 32-bit words, and (2) 32-bit words may contain some extra bits that do not belong to the integers of interest.
3. Aligning integers on bit boundaries, which may require shifting several integers to the right. Because SSE4 lacks a SIMD shift that has four different shift amounts, this step is simulated via a SIMD multiplication by four different integers using the SSE4 instruction `pmullud` followed by a vectorized right shift.
4. Zeroing bits that does not belong to the integers of interest. This requires a mask operation.
5. Storing the target buffer.

We compare experimentally vertical and horizontal bit packing in § 6.3.

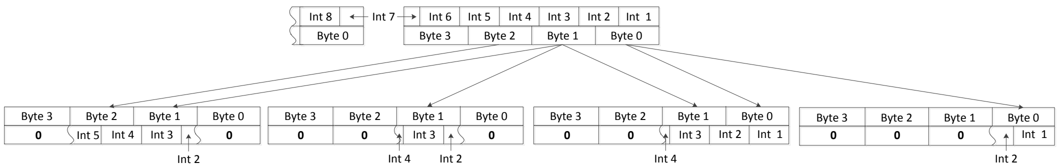


Figure 7. One step of simultaneous decoding of four 5-bit integers that are stored successively (as opposed to the interleaved data layout described by Fig. 3). These integers are copied to four 32-bit words using the shuffle operation `pshufb`. The locations in source and target buffers are indicated by arrows. Curvy lines are used to denote integers that cross byte boundaries in the source buffer. Hence, they are copied only partially. The boldface zero values represent the bytes zeroed by the shuffle instruction. Note that some source bytes are copied to multiple locations.

4.7. *Binary Packing with variable-length blocks*

Binary packing uses fixed-length blocks. Naturally, we could also vary the length of the blocks, in order to improve compression rate and decoding speed. It was first proposed by Deveaux et al. [36] who reported compression gains (15–30%). Delbru et al. [35] also implemented two such adaptive solutions, AFOR-2 and AFOR-3. AFOR-2 picks blocks of length 8, 16, 32 whereas AFOR-3 adds a special processing for the case where we have successive integers. To determine the best configuration of blocks, they pick 32 integers and try various configurations (1 block of 32 integers, 2 blocks of 16 integers and so on). Silvestri and Venturini [18] proposed two variable-length schemes, and we selected their fastest version (henceforth VSEncoding). Unlike AFOR-2 and AFOR-3, VSEncoding optimizes the block length using dynamic programming over blocks of lengths 1–14, 16, 32. Though Delbru et al. [35] did not compare their schemes with VSEncoding, they wrote that *given its almost identical nature, we can expect the results to be very close*.

4.8. *Patched coding*

Binary packing over long blocks (e.g., thousands of integers) might compress poorly because it is sensitive to outliers: a single large value forces an increase of the bit width on all other integers. For example, the integers $1, 4, 255, 4, 3, 12, 101$ can be stored using binary packing using 8 bits per integer for a total of $8 \times 8 = 64$ bits. However, the same sequence with one large value, e.g., $1, 4, 255, 4, 3, 12, 4294967295$ is not longer so compressible: $32 \times 8 = 256$ bits are required.

To alleviate this problem, Zukowski et al. [21] proposed *patching*: we use a small bit width b for all integers, but store exceptions (values greater than or equal to 2^b) in a separate location. They called this approach PFOR. (It is sometimes written PFD [37], PFor or PForDelta when used in conjunction with delta coding.) To determine the best bit width b during decoding, a sample of at most 2^{16} integers is created. Then, this sample is virtually compressed using various bit widths until the best compression rate is achieved.

In practice, to accelerate the computation, we can construct a histogram, recording how many integers have a given integer logarithm ($\lceil \log_2 x + 1 \rceil$). A single bit width is used for an entire page (e.g., 2^{23} integers).

The data is coded in blocks of 128 integers, with a *separate* storage array for the exceptions. The blocks are coded using bit packing. We either pack the integer value itself when the value is regular ($< 2^b$), or an integer offset pointing to the next exception in the block of 128 integers. The bit-packed blocks are preceded by a 32-bit word containing two markers. The first marker indicates the location of the first exception in the block of 128 integers, and the second marker indicates the location of this first exception value in the array of exceptions (exception table).

Effectively, exception locations are stored using a linked list: we first read the location of the first exception, then going to this location we find an offset from which we retrieve the location of the next exception, and so on. If the bit width b is too small to store an offset value, that is, if the offset is greater or equal than 2^b, we have to create a *compulsory* exception in-between. The location of

Table III. Overview of the patched coding schemes: Only PFOR and PFOR2008 generate compulsory exceptions and use the same bit width b per page. Only NewPFD and OptPFD store exceptions on a per block basis. We implemented all schemes with 128 integers per block and a page size of at least 2^{16} integers.

	compulsory	bit width	exceptions	compressed exceptions
PFOR [21]	yes	per page	per page	no
PFOR2008 [20]	yes	per page	per page	8, 16, 32 bits
NewPFD/OptPFD [7]	no	per block	per block	Simple-16
FastPFOR	no	per block	per page	binary packing
SIMD-FastPFOR	no	per block	per page	vectorized bin. pack.
SimplePFOR	no	per block	per page	Simple-8b

the exception values themselves are found by incrementing the location of the first exception value in the exception table.

When there are too many exceptions, these exception tables may overflow and it is necessary to start a new page: Zukowski et al. [21] used pages of 32 MB. In our own experiments, we partition large arrays into arrays of at most 2^{16} integers (see § 6.2) so a single page is used in practice.

PFOR [21] does not compress the exception values. In an attempt to improve the compression, Zhang et al. [20] proposed to store the exception values using either 8, 16, or 32 bits. We implemented this approach (henceforth PFOR2008). (See Table III.)

Nevertheless, the compression rates of PFOR and PFOR2008 are relatively modest (see § 6). For example, we found that they fare worse than binary packing (BP32). To get better compression, Yang et al. [7] proposed two new schemes called NewPFD and OptPFD. (NewPFD is sometimes called NewPFOR [38, 39] whereas OptPFD is also known as OPT-P4D [18].) Instead of using a uniform bit width b, they use the same bit width per a block of 128 integers. They avoid wasteful compulsory exceptions: instead of storing exception offsets in the bit packed blocks, they store the first b bits of the exceptional integer value. The $32 - b$ higher bits of the exception values (as well as their locations) are compressed using Simple-16 for each block of 128-integers. (We tried replacing Simple-16 with Simple-8b but we found no benefit.)

Each block of 128 coded integers is preceded by a 32-bit word used to store the bit width, the number of exceptions, and the storage requirement of the compressed exception values in 32-bit words. NewPFD determines the bit width b by picking the smallest value of b such that we do not have more than 10% of the integers as exceptions. OptPFD picks the value of b maximizing the compression. To accelerate the processing, the bit width is chosen among the integer values 0–16, 20 and 32.

Ao et al. [37] also proposed a version of PFOR called ParaPFD. It has worse compression rate than NewPFD or PFOR but it is designed for fast execution on GPUs.

5. NOVEL SCHEMES: SIMD-FASTPFOR, FASTPFOR AND SIMPLEPFOR

One of the key step with patching schemes NewPFD and OptPFD is to determine the best bit width in each block. In particular, the process used by OptPFD might be computationally expensive. We propose two new schemes, FastPFOR and SimplePFOR, that aim to optimize this encoding speed.

Instead of compressing the exceptions on a per block approach like NewPFD and OptPFD, FastPFOR and SimplePFOR store the exceptions as in the original PFOR scheme, on a per page basis. For each block, we keep the number of bits actually used, the maximum number of bits any actual value may use, a counter indicating the number of exceptions and the exception locations as integers in $[0, 127]$. This information is stored in an array of 8-bit integers. The difference between the bit width used and the maximal bit width is used to estimate the cost of storing an exception, together with the fact that we store exception locations using 8 bits. We only store the number of exceptions when this difference is greater than zero. For coding each block of 128 integers, we first

build a histogram that tells us how many integers have a given integer logarithm value. From this histogram, we can quickly determine the value b that minimizes the expected storage.

- In the SimplePFOR scheme, we collect all exceptions and compress them using Simple-8b.
- In the FastPFOR scheme, we store exceptions in one of 32 arrays, one for each possible bit width (from 1 to 32). When encoding a block, the difference between the maximal bit width and b determines in which array the exceptions are stored. Each of the 32 array is then bit packed using the corresponding bit width. Arrays are padded so that their length is a multiple of 32 integers.

During decoding, the exceptions are first decoded in bulk. To ensure that we do not overwhelm the CPU cache, we process the data in pages of 2^{16} integers. We then unpack the integers and apply patching on a per block basis.

Though SimplePFOR and FastPFOR are similar in design to NewPFD and OptPFD, we find that they offer better coding and decoding speed. It is an indication that compressing and decompressing exceptions in bulk might be faster.

We also designed a new scheme, SIMD-FastPFOR, that is identical to FastPFOR except that it relies exclusively on vectorized bit packing (including encoding of exception values). The compression rate is slightly diminished for two reasons:

- The 32 exception arrays are padded so that their length is a multiple of 128 integers, instead of 32 integers.
- We insert some padding prior to storing bit packing data so that alignment on 128-bit boundaries is preserved.

This padding adds an overhead of about 0.3–0.4 bit per integer (see Table V).

6. EXPERIMENTS

The goal of our experiments is to evaluate the best known integer encoding methods. The first series of our test in § 6.4 is based on synthetic data sets first presented by Anh and Moffat [19]: ClusterData and Uniform. They have the benefit that they can be quickly implemented, thus helping reproducibility. We then confirm our results in § 6.5 using large realistic data sets based on TREC collections ClueWeb09 and GOV2.

6.1. Hardware

We carried out our experiments on a Linux server equipped with Intel Core i7 2600 (3.40 GHz, 8192 KB of L3 CPU cache) and 16 GB of RAM. The DDR3-1333 RAM has a transfer rate of ≈20,000 MB/s or ≈5300 mis. According to our tests, it can copy arrays at a rate of 2270 mis with the C function `memcpy`.

6.2. Software

We implemented our algorithms in C++ using GNU GCC 4.7. We use the optimization flag `-O3`. Because the varint-G8IU scheme requires SSSE3 instructions, we had to add the flag `-mssse3`. When compiling our implementation of Willhalm et al. [22] bit unpacking, we had to use the flag `-msse4.1` since it requires SSE4 instructions. Our complete source code is available online.†

Following Stepanov et al. [9], we compute speed based on the wall-clock in-memory processing. Wall-clock times include the time necessary for *delta coding and decoding*. During our tests, we do not retrieve or store data on disk: it is impossible to decode billions of integers per second when they are kept on disk.

†https://github.com/lemire/FastPFOR

Arrays containing more than 2^{16} integers (256 KB) are broken down into smaller chunks. Each chunk is decoded into two passes. In the first pass, we decompress deltas and store each delta value using a 32-bit word. In the second pass, we carry out an in-place computation of prefix sums. As noted in § 2, this approach greatly improves data locality and leads to an almost twofold improvement in decoding speed for the fastest schemes.

Our implementation of VSEncoding, NewPFD, and OptPFD is based on software published by Silvestri and Venturini [18]. They report that their implementation of OptPFD was validated against an implementation provided by the original authors [7]. We implemented varint-G8IU from Stepanov et al. [9] as well as Simple-8b from Anh and Moffat [19]. We also implemented the original PFOR scheme from Zukowski et al. [21] as well as its successor PFOR2008 from Zhang et al. [20]. Zukowski et al. made a distinction between PFOR and PFOR-Delta: we effectively use FOR-Delta since we apply PFOR to deltas.

Some schemes compress data in blocks of fixed length (e.g., 128 integers). We compress the remainder using Variable Byte as in Zhang et al. [20]. In our tests, most arrays are sufficiently large (compared to the block size). Thus, replacing Variable Byte by another scheme would make no or little difference.

Speeds are reported in millions of 32-bit integers per second (mis). Stepanov et al. report a speed of 1059 mis over the TREC GOV2 data set for their best scheme varint-G8IU. We got a similar speed (1300 mis).

VSEncoding, FastPFOR, and SimplePFOR use buffers during compression and decompression proportional to the size of the array. VSEncoding uses a persistent buffer of over 256 KB. We implemented SIMD-FastPFOR, FastPFOR, and SimplePFOR with a persistent buffer of slightly more than 64 KB. PFOR, PFOR2008, NewPFD, and OptPFD are implemented using persistent buffers proportional to the block size (128 integers in our tests): less than 512 KB in persistent buffer memory are used for each scheme. Both PFOR and PFOR2008 use pages of 2^{26} integers or 256 MB. During compression, PFOR, PFOR2008, SIMD-FastPFOR, FastPFOR, and SimplePFOR use a buffer to store exceptions. These buffers are limited by the size of the pages and they are released immediately.

The implementation of VSEncoding by Silvestri and Venturini [18] uses some SSE2 instructions through assembly during bit unpacking. Varint-G8IU makes explicit use of SSSE3 instructions through GCC intrinsics whereas SIMD-FastPFOR and SIMD-BP128 similarly use SSE2 instructions. Several schemes benefit from the use of the x86 assembly instruction `bsr` for the computation of the integer logarithm.

Though we tested vectorized delta coding with all schemes, we only report results for schemes that make explicit use of SIMD instructions (SIMD-FastPFOR, SIMD-BP128, and varint-G8IU). To ensure fast vector processing, we align all initial pointers on 16-byte boundaries.

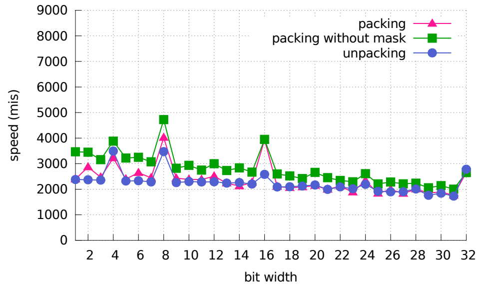
6.3. Computing bit packing

We implemented bit packing using hand-tuned functions as originally proposed by Zukowski et al. [21]. Given a bit width b, a sequence of K unsigned 32-bit integers are coded to $\lceil Kb/32 \rceil$ integers. In our tests, we used $K = 32$ for the regular version, and $K = 128$ for the vectorized version.

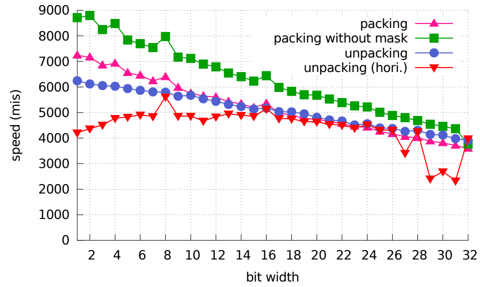
Fig. 8 illustrates the speed at which we can pack and unpack integers using blocks of 32 integers. In some schemes, it is known that all integers are no larger than $2^b - 1$, while in patching schemes there are exceptions, i.e., integers larger than or equal to 2^b. In the latter case, we enforce that integers are smaller than 2^b through the application of a mask. This operation slows down compression.

We can pack and unpack much faster when the number of bits is small because less data needs to be loaded in the CPU. We can pack and unpack faster when the bit width is 4, 8, 16, 24 or 32. Packing and unpacking with bit widths of 8 and 16 is especially fast.

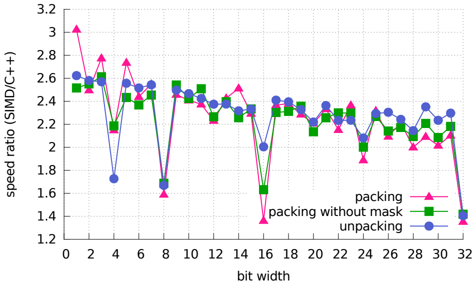
The vectorized version (Fig. 8b) is roughly twice as fast as the scalar version. We can unpack integers have a bit width of 8 or less at a rate of ≈6000 mis. However, it carries the implicit constraint that integers must be packed and unpacked in blocks of at least 128 integers with the same bit width. Packing is slightly faster when the bit width is 8 or 16.



(a) Optimized but portable C++



(b) Vectorized with SSE2 instructions



(c) Ratio vectorized/non-vectorized

Figure 8. Wall-clock speed in millions of integers per second for bit packing and unpacking. We use small arrays (256 KB) to minimize cache misses. When packing integers that do not necessarily fit in b bits, (as required in patching schemes), we must apply a mask which slows down packing by as much as 30%.

In Fig. 8b only, we reported the unpacking speed when using an horizontal data layout with a SIMD implementation based on Willhalm et al. [22] (see § 3 and 4.6.1). When the bit widths range from 16 to 26, this speed is the same as ours. For small (< 8) or large (> 27) bit widths, our approach based on a vertical layout is preferable as it is up to 70% faster.

We also experimented with the cases where we pack fewer integers ($K = 8$ or $K = 16$). However, it is slower and a few bits remain unused ($\lceil Kb/32 \rceil 32 - Kb$).

6.4. *Synthetic data sets*

We used successively the ClusterData and the Uniform model from Anh and Moffat [19]. In the Uniform model, integers follow a uniform distribution whereas in the ClusterData model, integer values are likely to cluster. That is, we are more likely to get long sequences of similar values. The goal of the ClusterData model is to simulate more realistically data encountered in practice. We expect data obtained from the ClusterData model to be more compressible.

We generated data sets of random integers in the range $[0, 2^{29})$. In the first pass, we generated 2^{10} short arrays containing 2^{15} integers each. The average difference between successive integers within an array is thus $2^{29-15} = 2^{14}$. We expect the compressed data to use at least 14 bits/int. In the second pass, we generated a single long array of 2^{25} integers. In this case, the average distance between successive integers is 2^4: we expect the schemes to use at least 4 bits/int.

The results are given in Table IV (schemes with a * by their name, e.g., SIMD-FastPFOR*, use vectorized delta coding). Over short arrays, we see little compression as expected. There is also a relatively little difference in compression rate between Variable Byte and a more space-efficient alternative such as FastPFOR. However, speed differences are large: the decoding speed ranges from 220 mis for Variable Byte to 2500 mis for SIMD-BP128*.

For long arrays, there is a greater difference between the compression rates. The schemes with the best compression rates are SIMD-FastPFOR, FastPFOR, SimplePFOR, Simple-8b, OptPFD. Among those, SIMD-FastPFOR is the clear winner in terms of decoding speed. The good

compression rate of OptPFD comes at a price: it has one of the worst encoding speeds. In fact, it is 20–50 times slower than SIMD-FastPFOR during encoding.

Though they differ significantly in implementation, FastPFOR, SimplePFOR, and SIMD-FastPFOR have equally good compression rates. All three schemes have similar decoding speeds, but SIMD-FastPFOR decodes integers much faster than FastPFOR and SimplePFOR.

In general, encoding speed vary significantly, but binary packing schemes are the fastest, especially when they are vectorized. Better implementations could possibly help reduce this gap.

The version of SIMD-BP128 using vectorized delta coding (written SIMD-BP128*) is always 400 mis faster during decoding than any other alternative. Though it does not always offer the best compression rate, it always matches the compression rate of Variable Byte.

The difference between using vectorized delta coding and regular delta coding could amount to up to 2 bits per integer. For example, SIMD-BP128* only uses about one extra bit per integer when compared with SIMD-BP128. The cost of binary packing is determined by the largest delta in a block: increasing the average size of the deltas by a factor of 4 does not necessarily lead to a fourfold increase in the expected largest integer (in a block of 128 deltas).

Compared to our novel schemes, performance of varint-G8IU is unimpressive. However, variant-G8IU is about 60% faster than Variable Byte while providing a similar compression rate. It is also faster than Simple-8b, though Simple-8b has a better compression rate. The version with vectorized delta coding (written varint-G8IU*) has poor compression over the short arrays compared with the regular version (varint-G8IU). Otherwise, on long arrays, varint-G8IU* is significantly faster (from 1300 mis to 1600 mis) than varint-G8IU while compressing just as well.

There is little difference between PFOR and PFOR2008 except that PFOR offers a significantly faster encoding speed. Among all the schemes taken from the literature, PFOR and PFOR2008 have the best decoding speed in these tests: they use a single bit width for all blocks, determined once at the beginning of the compression. However, they are dominated in all metrics (coding speed, decoding speed and compression rate) by SIMD-BP128 and SIMD-FastPFOR.

For comparison, we tested Google Snappy (version 1.0.5) as a delta compression technique. Google Snappy is a freely available library used internally by Google in its database engines [14]. We believe that it is competitive with other fast generic compression libraries such as zlib or LZO. For short ClusterData arrays, we got a decoding speed of 340 mis and almost no compression (29 bits/int.). For long ClusterData arrays, we got a decoding speed of 200 mis and 14 bits/int. Overall, Google Snappy has about half the compression rate of SIMD-BP128* while being an order of magnitude slower.

6.5. *Realistic data sets*

For more realistic data sets, we used posting lists obtained from two TREC Web collections. Our data sets include only document identifiers, but not positions of words in documents. That is, a posting list of a word is an array of document identifiers where the word occurs.

On the one hand, we used a posting list collection built from GOV2 data set by Silvestri and Venturini [18]. The GOV2 is a crawl of the `.gov` sites and contains 25 million HTML, text, and PDF documents (the latter are converted to text).

On the other hand, we used posting list collection extracted from the ClueWeb09 (Category B) data set [40]. This data set is a more realistic HTML collection of about 50 million crawled HTML documents, mostly in English. The ClueWeb09 collection represents postings for one million most frequent words. Common stop words were excluded and different grammar forms of words were conflated. Documents were enumerated in the order they appear in source files, i.e., they were not reordered. Unlike GOV2, the ClueWeb09 crawl is not limited to any specific domain. Uncompressed, the posting lists from GOV2 and ClueWeb09 use 20 GB and 50 GB respectively.

We decomposed these data sets according to the array length, storing all arrays of lengths 2^K to $2^{K+1}-1$ consecutively. We applied delta coding on the arrays $(x_1, x_2, \ldots \to x_1, x_2 - x_1, \ldots)$ and computed Shannon entropy $(\sum_i p(y_i) \log_2 p(y_i))$ over the set of integers produced (deltas plus initial values). We use a frequentist interpretation of Shannon entropy: the probability $p(y_i)$ of the integer value y_i is the number of occurrences of y_i divided by the number of integers. As Fig. 9

Table IV. Coding and decoding speed in millions of integers per second over synthetic data sets, together with number of bits per 32-bit integer. Results are given using two significant digits. Schemes with a $\star$ by their name use vectorized delta coding.

(a) ClusterData: Short arrays / (b) Uniform: Short arrays

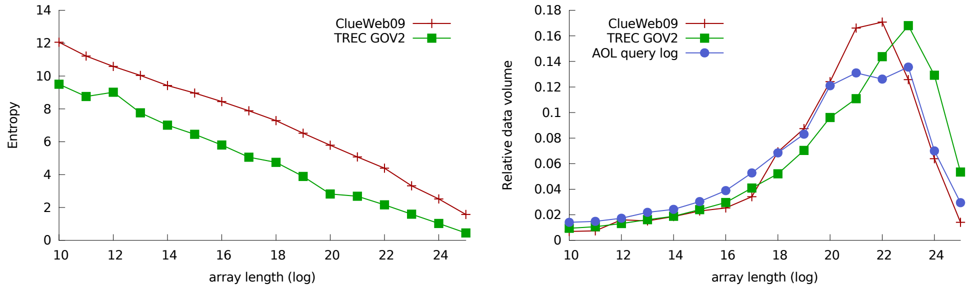
	(a) coding	(a) decoding	(a) bits/int	(b) coding	(b) decoding	(b) bits/int
SIMD-BP128$\star$	**1700**	**2500**	17	**1600**	**2000**	18
SIMD-FastPFOR$\star$	380	2000	16	360	1800	18
SIMD-BP128	1000	1800	16	1100	1600	17
SIMD-FastPFOR	300	1400	**15**	330	1400	**16**
PFOR	350	1200	18	370	1400	17
PFOR2008	280	1200	18	280	1400	17
SimplePFOR	300	1100	**15**	330	1200	**16**
FastPFOR	300	1100	**15**	330	1200	**16**
BP32	790	1100	**15**	840	1200	17
NewPFD	66	1100	16	64	1300	17
varint-G8IU$\star$	160	910	23	140	650	25
varint-G8IU	150	860	18	170	870	18
VSEncoding	10	720	16	10	690	18
Simple-8b	260	690	16	260	540	18
OptPFD	5.1	660	15	4.6	1100	17
Variable Byte	300	270	17	240	220	19

(c) ClusterData: Long arrays / (d) Uniform: Long arrays

	(c) coding	(c) decoding	(c) bits/int	(d) coding	(d) decoding	(d) bits/int
SIMD-BP128$\star$	**1800**	**2800**	7.0	**1900**	**2600**	8.0
SIMD-FastPFOR$\star$	440	2400	6.8	380	2200	7.6
SIMD-BP128	1100	1900	6.0	1100	1800	7.0
SIMD-FastPFOR	320	1600	5.4	340	1600	6.4
varint-G8IU$\star$	270	1600	9.1	270	1600	9.0
PFOR	360	1300	6.1	360	1300	7.3
PFOR2008	280	1300	6.1	280	1300	7.3
BP32	840	1300	5.8	810	1200	6.7
FastPFOR	320	1200	5.4	330	1200	6.3
SimplePFOR	320	1200	**5.3**	330	1200	6.3
varint-G8IU	230	1300	9.0	230	1300	9.0
NewPFD	120	970	5.5	110	1000	6.5
Simple-8b	360	890	5.6	370	940	6.4
VSEncoding	9.8	790	6.4	9.9	790	7.2
OptPFD	17	750	5.4	15	740	**6.2**
Variable Byte	880	830	8.1	930	860	8.0

shows, longer arrays are more compressible. There are differences in entropy values between two collections (ClueWeb09 has about two extra bits, see Fig. 9a), but these differences are much smaller than those among different array sizes. Fig. 9b shows the distribution of arrays per length as well as respective entropy values.

6.5.1. Results over different array lengths We present results per array length for selected schemes in Fig. 10. We see in Figs. 10b and 10f that all schemes compress the deltas within a factor of two of Shannon entropy for short arrays. For long arrays however, the compression rate (as compared to Shannon entropy) becomes worse for all schemes. Yet many of them manage to remain within a factor of three of Shannon entropy.



(a) Shannon entropy of the deltas (b) Data distribution

Figure 9. Description of the posting-list data sets

Integer compression schemes are better able to compress close to Shannon entropy over ClueWeb (see Fig. 10f) than over GOV2 (see Fig. 10b). For example, SIMD-FastPFOR, Simple-8b, and OptPFD are within a factor of two of Shannon entropy over ClueWeb for all array lengths, whereas they all exceed three times Shannon entropy over GOV2 for the longest arrays. Similarly, varint-G8IU, SIMD-BP128⋆, and SIMD-FastPFOR⋆ remain within a factor of six of Shannon entropy over ClueWeb09 but they all exceed this factor over GOV2 for long arrays. In general, it might be easier to compress data close to the entropy when the entropy is larger.

We get poor results with varint-G8IU over long arrays: if an array has more than 2^{20} elements, an average code size is 9–10 bits (see Figs. 10a and 10e). We do not find this surprising, because variant-G8IU is a modification of a Variable Byte encoding and, thus, cannot store deltas using a code shorter than one byte. At the same time, integers in long arrays tend to be smaller than 256.

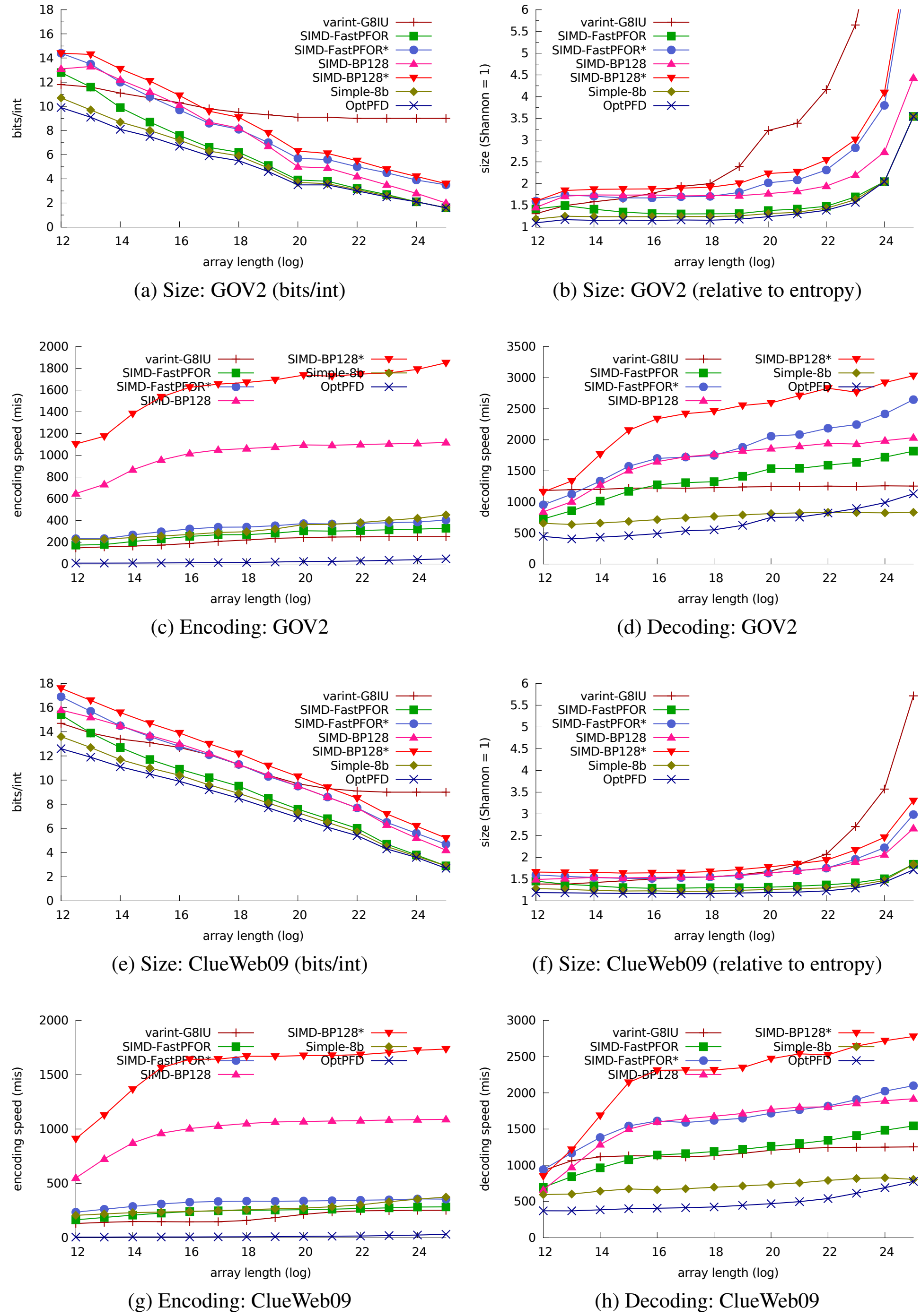
We see in Figs. 10c and 10g that both SIMD-BP128 and SIMD-BP128⋆ have a significantly better encoding speed, irrespective of the array length. The opposite is true for OptPFD: it is much slower than the alternatives.

Examining the decoding speed as a function of array length (see Figs. 10cand 10g), we see that several schemes have a significantly worse decoding speed over short arrays, but the effect is most pronounced for the new schemes we introduced (SIMD-FastPFOR, SIMD-FastPFOR⋆, SIMD-BP128, and SIMD-BP128⋆). Meanwhile, varint-G8IU and Simple-8b have a decoding speed that is less sensitive to the array length.

Varint-G8IU is one of the fastest methods available and it might be well suited for short arrays. Indeed, it offers both a better speed and a better compression rate than most alternatives when arrays have length smaller than 2^{14} integers.

6.5.2. Aggregated results Not all posting lists are equally likely to be retrieved by the search engine. As observed by Stepanov et al. [9], it is desirable to account for different term distributions in queries. Unfortunately, we do not know of an ideal approach to this problem. Nevertheless, to model more closely the performance of a major search engine, we used the AOL query log data set as a collection of query statistics [41, 42]. It consists in about 20 million web queries collected from 650 thousand users over three months: queries repeating within a single user session were ignored. When possible (in about 90% of all cases), we matched the query terms with posting lists in the ClueWeb09 data set and obtained term frequencies (see Fig. 9b). This allowed us to estimate how often a posting list of length between 2^K to $2^{K+1}-1$ is likely to be retrieved for various values of K. This gave us a weight vector that we use to aggregate our results.

We present aggregated results in Table V. The results are generally similar to what we obtained with synthetic data. The newly introduced schemes (SIMD-BP128⋆, SIMD-FastPFOR⋆, SIMD-BP128, SIMD-FastPFOR) still offer the best decoding speed. We find that varint-G8IU⋆ is much faster than varint-G8IU (1500 mis vs. 1300 mis over GOV2) even though the compression rate is the same with a margin of 10%. PFOR and PFOR2008 offer a better compression than



(a) Size: GOV2 (bits/int)

(b) Size: GOV2 (relative to entropy)

(c) Encoding: GOV2

(d) Decoding: GOV2

(e) Size: ClueWeb09 (bits/int)

(f) Size: ClueWeb09 (relative to entropy)

(g) Encoding: ClueWeb09

(h) Decoding: ClueWeb09

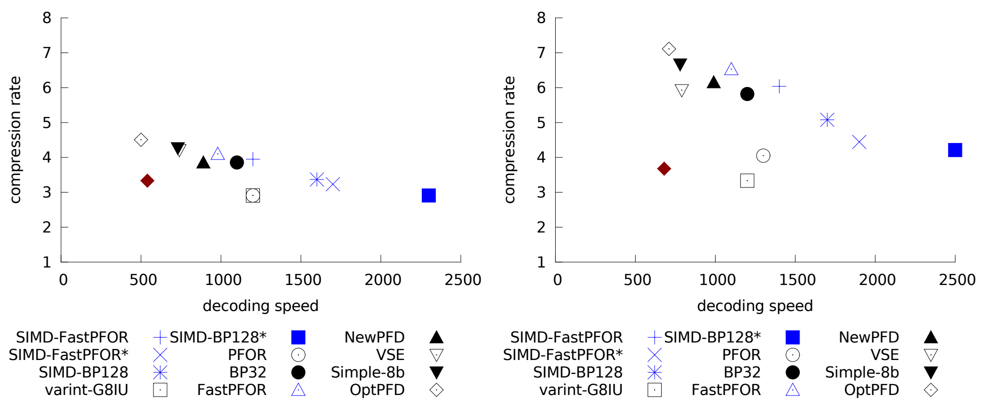
Figure 10. Experimental comparison of competitive schemes on Clueweb09 and GOV2.

varint-G8IU* but at a reduced speed. However, we find that SIMD-BP128 is preferable in every way to varint-G8IU*, varint-G8IU, PFOR, and PFOR2008.

For some applications, decoding speed and compression rates are the most important metrics. Whereas elsewhere we report the number of bits per integer b, we can easily compute the compression rate as $32/b$. We plot both metrics for some competitive schemes (see Fig. 11).

Table V. Experimental results Coding and decoding speeds are given in millions of 32-bit integers per second. Averages are weighted based on AOL query logs.

	(a) ClueWeb09			(b) GOV2		
	coding	decoding	bits/int	coding	decoding	bits/int
SIMD-BP128⋆	**1600**	**2300**	11	**1600**	**2500**	7.6
SIMD-FastPFOR⋆	330	1700	9.9	350	1900	7.2
SIMD-BP128	1000	1600	9.5	1000	1700	6.3
varint-G8IU⋆	220	1400	12	240	1500	10
SIMD-FastPFOR	250	1200	8.1	290	1400	5.3
PFOR2008	260	1200	10	250	1300	7.9
PFOR	330	1200	11	310	1300	7.9
varint-G8IU	210	1200	11	230	1300	9.6
BP32	760	1100	8.3	790	1200	5.5
SimplePFOR	240	980	7.7	270	1100	4.8
FastPFOR	240	980	7.8	270	1100	4.9
NewPFD	100	890	8.3	150	1000	5.2
VSEncoding	11	740	7.6	11	810	5.4
Simple-8b	280	730	7.5	340	780	4.8
OptPFD	14	500	**7.1**	23	710	**4.5**
Variable Byte	570	540	9.6	730	680	8.7



(a) ClueWeb09 (b) GOV2

Figure 11. Scatter plots comparing competitive schemes on decoding speed and compression rate weighted based on AOL query logs. We use VSE as a shorthand for VSEncoding. For reference, Variable Byte is indicated as a dark-red lozenge. The novel schemes (e.g., SIMD-BP128⋆) are identified with blue markers.

These plots suggest that the most competitive schemes are SIMD-BP128⋆, SIMD-FastPFOR⋆, SIMD-BP128, SIMD-FastPFOR, SimplePFOR, FastPFOR, Simple-8b, and OptPFD depending on how much compression is desired. Fig. 11 also shows that when decoding speeds higher than 1300 mis are required, we must choose between SIMD-BP128, SIMD-FastPFor⋆, and SIMD-BP128⋆.

Few research papers report encoding speed. Yet we find large differences: for example, VSEncoding and OptPFD are two orders of magnitude slower during encoding than our fastest schemes. If the compressed arrays are written to slow disks in a batch mode, such differences might be of little practical significance. However, for memory-based databases and network applications, slow encoding speeds could be a concern. Our SIMD-BP128 and SIMD-BP128⋆ schemes have especially fast encoding.

Table VI. Average compression rates and speeds in millions of 32-bit integers per second over all arrays of two data sets. These averages are **not** weighted according to the AOL query logs.

	(a) ClueWeb09			(b) GOV2		
	coding	decoding	bits/int	coding	decoding	bits/int
SIMD-BP128*	**1600**	**2400**	9.7	**1600**	**2500**	7.4
SIMD-FastPFOR*	340	1700	9.0	350	1900	6.9
SIMD-BP128	1000	1700	8.7	1000	1800	6.1
varint-G8IU*	230	1400	12	250	1500	10
SIMD-FastPFOR	260	1300	7.2	290	1400	5.1
PFOR2008	260	1300	9.6	250	1300	7.6
PFOR	330	1300	9.6	310	1300	7.6
varint-G8IU	220	1200	10	230	1200	9.6
BP32	770	1100	7.5	790	1200	5.3
SimplePFOR	250	1000	6.9	260	1100	4.7
FastPFOR	240	1000	6.9	270	1100	4.8
NewPFD	110	900	7.4	150	990	5.0
VSEncoding	11	760	6.9	11	790	5.4
Simple-8b	290	750	6.7	340	780	4.6
OptPFD	16	530	**6.4**	25	720	**4.4**
Variable Byte	630	600	9.2	750	700	8.6

Similarly to previous work [9, 18], in Table VI we report unweighted averages. The unweighted speed aggregates are equivalent to computing the average speed over all arrays—irrespective of their lengths. From the distribution of posting size logarithms in Fig. 9b, one may conclude that weighted results should be similar to unweighted ones. These observations are supported by data in Table VI: the decoding speeds and compression rates for both aggregation approaches differ by less than 15% with the weighted results presented in Table V.

We can compare the number of bits per integer in Table VI with an information theoretical limit. Indeed, Shannon entropy for the deltas of ClueWeb09 is 5.5 bits/int whereas it is 3.6 for GOV2. Hence, OptPFD is within 16% of the entropy on ClueWeb09 whereas it is within 22% of the entropy on GOV2. Meanwhile, the faster SIMD-FastPFOR is within 30% and 40% of the entropy for ClueWeb09 and GOV2. Our fastest scheme (SIMD-BP128*) compresses the deltas of GOV2 to twice the entropy. It does slightly better with ClueWeb09 ($1.8\times$).

7. DISCUSSION

We find that binary packing is both fast and space efficient. The vectorized binary packing (SIMD-BP128*) is our fastest scheme. It is true that it has a lesser compression rate compared to Simple-8b (by about 50%), but it is more than 3 times faster. Moreover, in the worst case, a slower binary packing scheme (BP32) incurred a cost of only about 1.2 bit per integer compared to the patching scheme with the best compression ratio (OptPFD) while being nearly as fast (within 10%) as the fastest patching scheme (PFOR).

Yet only few authors considered binary packing schemes or its vectorized variants in the recent literature:

- Delbru et al. [35] reported good results with a binary packing scheme similar to our BP32: in their experiments, it surpassed Simple-8b as well as a patched scheme (PFOR2008).
- Anh and Moffat [6] also reported good results with a binary packing scheme: in their tests, it was faster than either Simple-8b or PFOR2008 by at least 50%. As a counterpart, they reported that their binary packing scheme had a poorer compression.

- Schlegel et al. [26] proposed a scheme similar to SIMD-BP128. This scheme (called k-gamma) stores integer in a round-robin fashion (see § 4.6.1) like our SIMD-BP128 and SIMD-FastPFOR schemes. It essentially applies binary packing to tiny groups of integers (at most 4 elements). From our preliminary experiments, we learned that decoding integers in small groups is not efficient. This is also supported by results of Schlegel et al. [26]. Their fastest decoding speed, which does not include memorization of decoded integers, is only 1600 mis (Core i7-920, 2.67 Ghz).
- Willhalm et al. [22] used a vectorized binary packing like our SIMD-BP128, but with a sequential (horizontal) data layout instead of our interleaved (vertical) layout. The decoding algorithm relies on the shuffle instruction `pshufb`. Our experimental results suggest that our approach based on a vertical layout might be preferable (see Fig. 8a): our implementation of bit unpacking over a vertical layout is sometimes between 50% to 70% faster than our reimplementation over a horizontal layout based on the work of Willhalm et al. [22].
  This performance comparison depends on the quality of our software. Yet the speed of our reimplementation is comparable with the speed originally reported by Willhalm et al. [22, Fig. 11]: they report a speed of ≈3300 mis with a bit width of 6. In contrast, using our implementation of their algorithms, we got a speed above 4800 mis for the same bit width and a 20% higher clock speed.
  However, the approach described by Willhalm et al. might be more competitive on platforms with instructions for simultaneously shifting several values by different offsets (e.g., the `vpsrld` AVX2 instruction). Indeed, this must be otherwise emulated by multiplications by powers of two followed by shifting.

Vectorized bit-packing schemes are efficient: they encode/decode integers at speeds of 4000–8500 mis. Hence, the computation of deltas and prefix sums may become a major bottleneck. This bottleneck can be removed through vectorization of these operations (though at expense of poorer compression rates in our case). We have not encountered this approach in the literature: perhaps, because for slower schemes the computation of the prefix sum accounts for a small fraction of total running time. In our implementation, to ease comparisons, we have separated delta decoding from data decompression: an integrated approach could be faster in some cases. Moreover, we might be able improve the decoding speed and the compression rates with better vectorized algorithms. There might also be alternatives to data differencing, which also permit vectorization, such as linear regression [37].

In our results, the original patched coding scheme (PFOR) is bested on all three metrics (compression rate, coding and decoding speed) by a binary packing scheme (SIMD-BP128). Similarly, a more recent fast patching scheme (NewPFD) is generally bested by another binary packing scheme (BP32). Indeed, though the compression rate of NewPFD is up to 6% better on realistic data, NewPFD is at least 20% slower than BP32. Had we stopped our investigations there, we might have been tempted to conclude that patched coding is not a viable solution when decoding speed is the most important characteristic on desktop processors. However, we designed a new vectorized patching scheme SIMD-FastPFOR. It shows that patching remains a fruitful strategy even when SIMD instructions are used. Indeed, it is faster than the SIMD-based varint-G8IU while providing a much better compression rate (by at least 35%). In fact, on realistic data, SIMD-FastPFOR is better than BP32 on two key metrics: decoding speed and compression rate (see Fig. 11).

In the future, we may expect increases in the arity of SIMD operations supported by commodity CPUs (e.g., with AVX) as well as in memory speeds (e.g., with DDR4 SDRAM). These future improvements could make our vectorized schemes even faster in comparison to their scalar counterparts. However, an increase in arity means an increase in the minimum block size. Yet, when we increase the size of the blocks in binary packing, we also make them less space efficient in the presence of outlier values. Consider that BP32 is significantly more space efficient than SIMD-BP128 (e.g., 5.5 bits/int vs. 6.3 bits/int on GOV2).

Thankfully, the problem of outliers in large blocks can be solved through patching. Indeed, even though OptPFD uses the same block size as SIMD-BP128, it offers significantly better compression

(4.5 bits/int vs. 6.3 bits/int on GOV2). Thus, patching may be more useful for future computers—capable of processing larger vectors—than for current ones.

8. CONCLUSION

We have presented new schemes that are up to twice as fast as the previously best available schemes in the literature while offering competitive compression rates and encoding speed. This was achieved by vectorization of almost every step including delta decoding. To achieve both high speed and competitive compression rates, we introduced a new patched scheme that stores exceptions in a way that permits a vectorization (SIMD-FastPFOR).

In the future, we might seek to generalize our results over more varied architectures as well as to provide a greater range of tradeoffs between speed and compression rate. Indeed, most commodity processors support vector processing (e.g., Intel, AMD, PowerPC, ARM). We might also want to consider adaptive schemes that compress more aggressively when the data is more compressible and optimize for speed otherwise. One could also use workload-aware compression: frequently accessed arrays could be optimized for decoding speed whereas least frequently accessed data could be optimized for high compression rate.

ACKNOWLEDGEMENT

Our varint-G8IU implementation is based on code by M. Caron. V. Volkov provided better loop unrolling for delta coding. P. Bannister provided a fast algorithm to compute the maximum of the integer logarithm of an array of integers. We are grateful to N. Kurz for discussions on memory speed.

REFERENCES

1. Sebot J, Drach-Temam N. Memory bandwidth: The true bottleneck of SIMD multimedia performance on a superscalar processor. *Euro-Par 2001 Parallel Processing*, *Lecture Notes in Computer Science*, vol. 2150. Springer Berlin / Heidelberg, 2001; 439–447, doi:10.1007/3-540-44681-8_63.
2. Drepper U. What every programmer should know about memory. http://www.akkadia.org/drepper/cpumemory.pdf [Last checked August 2012.] 2007.
3. Westmann T, Kossmann D, Helmer S, Moerkotte G. The implementation and performance of compressed databases. *SIGMOD Record* September 2000; **29**(3):55–67, doi:10.1145/362084.362137.
4. Abadi D, Madden S, Ferreira M. Integrating compression and execution in column-oriented database systems. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, ACM: New York, NY, USA, 2006; 671–682, doi:10.1145/1142473.1142548.
5. Büttcher S, Clarke CLA. Index compression is good, especially for random access. *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, ACM: New York, NY, USA, 2007; 761–770, doi:10.1145/1321440.1321546.
6. Anh VN, Moffat A. Inverted index compression using word-aligned binary codes. *Information Retrieval* 2005; **8**(1):151–166, doi:10.1023/B:INRT.0000048490.99518.5c.
7. Yan H, Ding S, Suel T. Inverted index compression and query processing with optimized document ordering. *Proceedings of the 18th international conference on World wide web*, WWW '09, ACM: New York, NY, USA, 2009; 401–410, doi:10.1145/1526709.1526764.
8. Popov P. Basic optimizations: Talk at the YaC (Yet Another Conference) held by Yandex (in Russian). http://yac2011.yandex.com/archive2010/topics/ [Last checked Sept 2012.] 2010.
9. Stepanov AA, Gangolli AR, Rose DE, Ernst RJ, Oberoi PS. SIMD-based decoding of posting lists. *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, ACM: New York, NY, USA, 2011; 317–326, doi:10.1145/2063576.2063627.
10. Dean J. Challenges in building large-scale information retrieval systems: invited talk. *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, ACM: New York, NY, USA, 2009; 1–1, doi:10.1145/1498759.1498761.
11. Lemke C, Sattler KU, Faerber F, Zeier A. Speeding up queries in column stores: a case for compression. *Proceedings of the 12th international conference on Data warehousing and knowledge discovery*, DaWaK'10, Springer-Verlag: Berlin, Heidelberg, 2010; 117–129, doi:10.1007/978-3-642-15105-7_10.
12. Binnig C, Hildenbrand S, Färber F. Dictionary-based order-preserving string compression for main memory column stores. *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ACM: New York, NY, USA, 2009; 283–296, doi:10.1145/1559845.1559877.
13. Poess M, Potapov D. Data compression in Oracle. *VLDB'03, Proceedings of the 29th International Conference on Very Large Data Bases*, Morgan Kaufmann: San Francisco, CA, USA, 2003; 937–947.

14. Hall A, Bachmann O, Bssow R, Gänceanu S, Nunkesser M. Processing a trillion cells per mouse click. *Proceedings of the VLDB Endowment* 2012; **5**(11):1436–1446.
15. Raman V, Swart G. How to wring a table dry: entropy compression of relations and querying of compressed relations. *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, VLDB Endowment, 2006; 858–869.
16. Lemire D, Kaser O. Reordering columns for smaller indexes. *Information Sciences* June 2011; **181**(12):2550–2570, doi:10.1016/j.ins.2011.02.002.
17. Bjørklund TA, Grimsmo N, Gehrke J, Torbjørnsen O. Inverted indexes vs. bitmap indexes in decision support systems. *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, ACM: New York, NY, USA, 2009; 1509–1512, doi:10.1145/1645953.1646158.
18. Silvestri F, Venturini R. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, ACM: New York, NY, USA, 2010; 1219–1228, doi:10.1145/1871437.1871592.
19. Anh VN, Moffat A. Index compression using 64-bit words. *Software: Practice and Experience* 2010; **40**(2):131–147, doi:10.1002/spe.v40:2.
20. Zhang J, Long X, Suel T. Performance of compressed inverted list caching in search engines. *Proceedings of the 17th international conference on World Wide Web*, WWW '08, ACM: New York, NY, USA, 2008; 387–396, doi:10.1145/1367497.1367550.
21. Zukowski M, Heman S, Nes N, Boncz P. Super-scalar RAM-CPU cache compression. *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, IEEE Computer Society: Washington, DC, USA, 2006; 59–71, doi:10.1109/ICDE.2006.150.
22. Willhalm T, Popovici N, Boshmaf Y, Plattner H, Zeier A, Schaffner J. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* Aug 2009; **2**(1):385–394.
23. Zhou J, Ross KA. Implementing database operations using SIMD instructions. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, ACM: New York, NY, USA, 2002; 145–156, doi:10.1145/564691.564709.
24. Inoue H, Moriyama T, Komatsu H, Nakatani T. A high-performance sorting algorithm for multicore single-instruction multiple-data processors. *Software: Practice and Experience* Jun 2012; **42**(6):753–777, doi:10.1002/spe.1102.
25. Wassenberg J. Lossless asymmetric single instruction multiple data codec. *Software: Practice and Experience* 2012; **42**(9):1095–1106, doi:10.1002/spe.1109.
26. Schlegel B, Gemulla R, Lehner W. Fast integer compression using SIMD instructions. *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, ACM: New York, NY, USA, 2010; 34–40, doi:10.1145/1869389.1869394.
27. Witten IH, Moffat A, Bell TC. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1999.
28. Rice R, Plaunt J. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology* 1971; **19**(6):889 –897.
29. Elias P. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory* 1975; **21**(2):194–203.
30. Transier F, Sanders P. Engineering basic algorithms of an in-memory text search engine. *ACM Transactions on Information Systems* Dec 2010; **29**(1):2:1–2:37, doi:10.1145/1877766.1877768.
31. Moffat A, Stuiver L. Binary interpolative coding for effective index compression. *Information Retrieval* 2000; **3**(1):25–47, doi:10.1023/A:1013002601898.
32. Walder J, Krátký M, Bača R, Platoš J, Snášel V. Fast decoding algorithms for variable-lengths codes. *Information Sciences* Jan 2012; **183**(1):66–91, doi:10.1016/j.ins.2011.06.019.
33. Goldstein J, Ramakrishnan R, Shaft U. Compressing relations and indexes. *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, IEEE Computer Society: Washington, DC, USA, 1998; 370–379.
34. Ng WK, Ravishankar CV. Block-oriented compression techniques for large statistical databases. *IEEE Transactions on Knowledge and Data Engineering* Mar 1997; **9**(2):314–328, doi:10.1109/69.591455.
35. Delbru R, Campinas S, Tummarello G. Searching web data: An entity retrieval and high-performance indexing model. *Web Semantics* Jan 2012; **10**:33–58, doi:10.1016/j.websem.2011.04.004.
36. Deveaux JP, Rau-Chaplin A, Zeh N. Adaptive Tuple Differential Coding. *Database and Expert Systems Applications*, *Lecture Notes in Computer Science*, vol. 4653. Springer Berlin / Heidelberg, 2007; 109–119, doi:10.1007/978-3-540-74469-6_12.
37. Ao N, Zhang F, Wu D, Stones DS, Wang G, Liu X, Liu J, Lin S. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment* May 2011; **4**(8):470–481.
38. Baeza-Yates R, Jonassen S. Modeling static caching in web search engines. *Advances in Information Retrieval*, *Lecture Notes in Computer Science*, vol. 7224. Springer Berlin / Heidelberg, 2012; 436–446, doi:10.1007/978-3-642-28997-2_37.
39. Jonassen S, Bratsberg S. Intra-query concurrent pipelined processing for distributed full-text retrieval. *Advances in Information Retrieval*, *Lecture Notes in Computer Science*, vol. 7224. Springer Berlin / Heidelberg, 2012; 413–425, doi:10.1007/978-3-642-28997-2_35.
40. Boystov L. Clueweb09 posting list data set. http://boytsov.info/datasets/clueweb09gap/ [Last checked August 2012.] 2012.
41. Brenes DJ, Gayo-Avello D. Stratified analysis of AOL query log. *Information Sciences* May 2009; **179**(12):1844–1858, doi:10.1016/j.ins.2009.01.027.
42. Pass G, Chowdhury A, Torgeson C. A picture of search. *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, ACM: New York, NY, USA, 2006, doi:10.1145/1146847.1146848.

A. INFORMATION THEORETICAL BOUND ON BINARY PACKING

Consider arrays of n distinct sorted 32-bit integers. We can compress the deltas computed from such arrays using binary packing as described in § 4.6 (see Fig. 1). We want to prove that such an approach is reasonably efficient.

There are $\binom{2^{32}}{n}$ such arrays. Thus, by an information theoretical argument, we need at least $\log\binom{2^{32}}{n}$ bits to represent them. By a well known inequality, we have that $\log\binom{2^{32}}{n} \geq n\log\frac{2^{32}}{n}$. In effect, this means that we need at least $\log\frac{2^{32}}{n}$ bits/int.

Consider binary packing over blocks of B integers: e.g., for BP32 we have $B = 32$ and for SIMD-BP128 we have $B = 128$. For simplicity, assume that the array length n is divisible by B and that B is divisible by 32. Though our result also holds for vectorized delta coding (§ 2), assume that we use the common version of delta coding before applying binary packing. That is, if the original array is $x_1, x_2, x_3, \ldots$ ($x_i > x_{i-1}$ for all $i > 1$), we compress the integers $x_1, x_2 - x_1, x_3 - x_2, \ldots$ using binary packing.

For every block of B integers, we have an overhead of 8 bits to store the bit width b. This contributes $8n/B$ bits to the total storage cost. The storage of any given block depends also on the bit width for this block. In turn, the bit width is bounded by the logarithm of the difference between the largest and the smallest element in the block. If we write this difference for block i as Δ_i, the total storage cost in bits is

$$\frac{8n}{B} + \sum_{i=1}^{n/B} B\lceil\log(\Delta_i)\rceil \quad \leq \quad \frac{8n}{B} + n + B\log\left(\prod_{i=1}^{n/B}\Delta_i\right).$$

Because $\sum_{i=1}^{n/B}\Delta_i \leq 2^{32}$, we can show that the cost is maximized when $\Delta_i = 2^{32}B/n$. Thus, we have that the *total* cost in bits is smaller than

$$\begin{aligned}\frac{8n}{B} + n + B\log\left(\prod_{i=1}^{n/B}\frac{2^{32}B}{n}\right) \quad &= \quad \frac{8n}{B} + n + B\log\left(\frac{2^{32}B}{n}\right)^{n/B} \\ &= \quad \frac{8n}{B} + n + n\log\frac{2^{32}B}{n},\end{aligned}$$

which is equivalent to $8/B + 1 + \log B + \log\frac{2^{32}}{n}$ bits/int. Hence, in the worst case, binary packing is suboptimal by $8/B + 1 + \log B$ bits/int. Therefore, we can show that BP32 is 2-optimal for arrays of length less than 2^{25} integers: its storage cost is no more than twice the information theoretical limit. We also have that SIMD-BP128 is 2-optimal for arrays of length 2^{23} or less.